18. Статична и динамична памет. Указатели и псевдоними

Проф. д-р Емил Хаджиколев

1. Обща представа за управление на паметта на програмата.
2. Достъп до данните – по стойност и по адрес
3. Указатели
4. Псевдоними

Концепции за управление на паметта на програмата

- **Статичната памет за програмата**.
- **Памет за стек** (stack memory) – **част от статичната памет**, служеща за изпълнение на функциите, която се управлява динамично.
- **Динамична памет** (хийп – heap memory) за данни.

- **Данните в статичната памет (и стека) се достъпват чрез асоциирани с тях имена на променливи.**
- **Данните в хийпа се достъпват/реферират чрез адрес**, който се задава като стойност на променлива, наричана указател, референция и др. в различни езици за програмиране.

Статичната памет за програмата

- Заделя се по време на компилацията;
- Съществува през цялото изпълнение на програмата;
- Включва глобални променливи и константи (декларирани извън функции) и статични променливи във функции.

Динамична памет

- **Управлява се** по време на изпълнение на програмата
  - **заделя се с оператор new в C++** и много други езици за програмиране;
  - **освобождава се**
    - **явно с оператор delete в езици като C++**
    - неявно чрез автоматизирани механизми като Garbage Collector в Java, C# и др.

Стек

- Статична памет за данни:
  - локални променливи (във функции) и
  - адреси на връщане от функции.
- Размерът на стека се определя по време на компилиране и обикновено е фиксиран.
- Управлява се динамично по време на изпълнение на програмата.
  - При извикване на функция се добавят данни към стека (push), а при излизане от функцията се премахват данни от стека (pop).
- При рекурсивно извикване на функции е възможно паметта за стека да не е пресметната точно (от компилатора, а и не винаги е възможно) и да се получи препълване на стека (грешка: stack overflow), при което програмата приключва неуспешно. При подобна опасност е добре да се търси итеративно решение.
- Програмата приключва след като стека се изпразни.

Предимства и недостатъци при статичната памет

- (+) данните се запазват по време на изпълнение на програмата, което позволява лесен достъп до тях.
- (-) размерът на (статичната) паметта на програмата е ограничен:
  - от компилатора (с настройки може да се променя в определени граници);
  - от процесора (при 32-битов процесор се адресират $2^{32} = 4$ гигабайта памет);
  - от изисквания на ОС;
  - ако оперативната памет е ограничена, ОС ще се затрудни с едновременно зареждане на приложения, които я превишават (при което ще използва виртуална памет върху твърдия диск, което ще забави общата производителност на системата) и др.;
- (-) не е възможно динамично увеличаване или намаляване на паметта според нуждите на програмата
  - напр. въпреки че при различни изпълнения на програмата е възможно да ни е необходим масив с различни размери, при статичен масив трябва предварително да предвидим той да е с максимален възможен брой елементи…

Предимства и недостатъци при динамичната памет

- (+) Гъвкаво управление на паметта – динамично заделяне на необходимата памет и освобождаването ѝ по време на изпълнение на програмата;
- (-) Риск от
  - неправилно управление на паметта, което създава изтичане/утечка на памет (memory leak);
  - неправилно освобождаване на памет, която се използва.

Тези рискове може да доведат до срив на програмата или увеличение на използваната памет.

Освобождаване на памет

- В C++ програмистът носи отговорност за правилното освобождаване на паметта.
  - Ако паметта, която няма да се използва, не се освобождава (например в края на функция, която се използва многократно), то заетата памет нараства, завзема памет, която е потребна и за други приложения, което в крайна сметка може да доведе до срив на ОС.
  - Ако паметта не се освободи (от програмиста) и не са възникнали проблеми от това, то при приключване на програмата, ОС автоматично ще освободи както статичната, така и динамичната памет за програмата.
- В други езици при механизми като Garbage Collector, за автоматично освобождаване на неизползваната динамична памет, също (лесно) могат да възникнат проблеми с паметта – неправилно заделяне (от програмиста на огромна памет), бавно (не своевременно) освобождаване на паметта и възникване на т.нар. OutOfMemoryError.

Памети за стек и хийп

- Памет за стек (stack):
  - Данните имат кратък жизнен цикъл – стекът съхранява кратковременно състоянието на локални променливи.
- Памет за хийп (heap):
  - Данните имат дълъг жизнен цикъл – хийпа се нарича дълготрайна памет.
  - Използва се за съхранение на данни, създадени по време на изпълнение на програмата.
  - Обектите в хийпа не са твърдо свързани с променливи. За достъп до обектите се използват променливи-указатели (които по време на изпълнение на програмата може да сочат към различни обекти, в различни моменти от време).

Статични и динамични данни

- Данни, за които се заделя памет в статичната памет (и стека) се наричат статични.
- Данни, за които се заделя памет в хийпа се наричат динамични.

- Без значение къде се намират, данните имат адрес, на който се намират и чрез който се работи с тях.

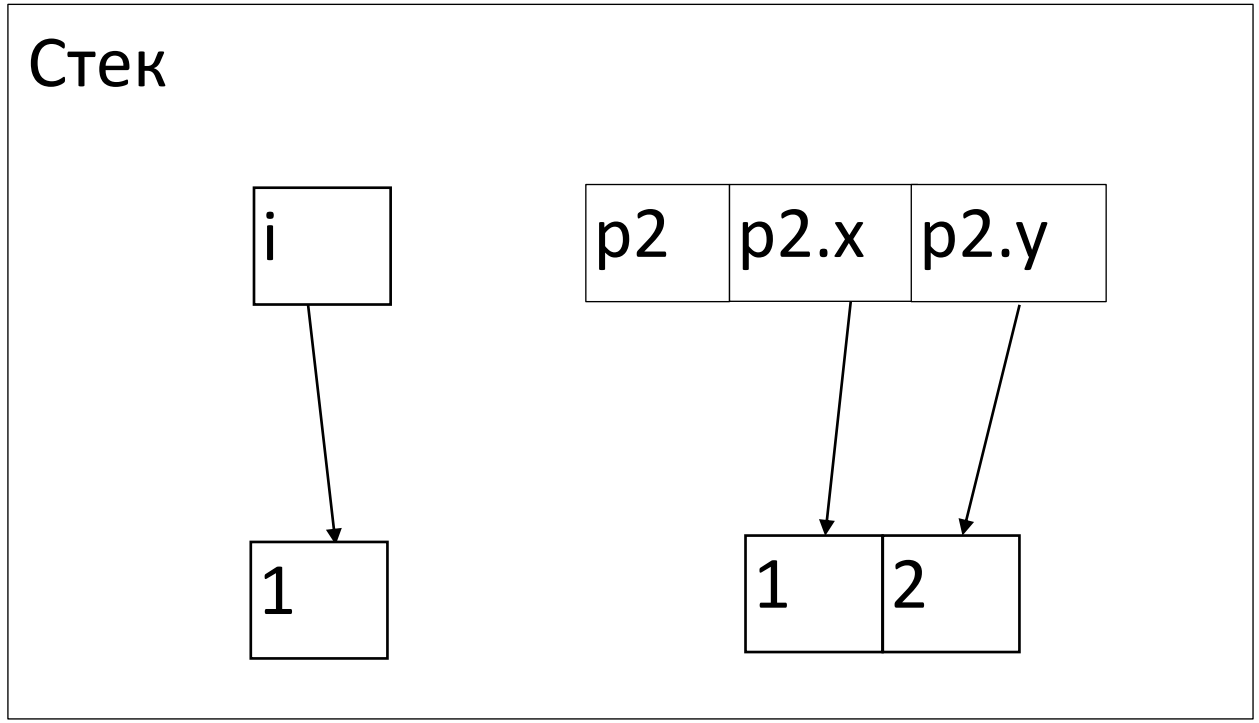
- За динамичните данни се заделя памет с оператор new, който връща начален адрес за данните и който адрес може да се присвои на реферираща променлива – указател.

Променливи

- Променливите се описват с име, тип и стойност.
- Чрез името се обръщаме към асоциираната с него стойност.

```
int i = 1; // статична променлива i в стека,
           // има стойност 1 в стека
Point2D p2;  // статична променлива p2 в стека
p2.x = 1; p2.y = 2; // полетата x и y също са
                    // в стека и те имат стойности
                    // 1 и 2 съответно, в стека
```

Представяне на променливите от примера



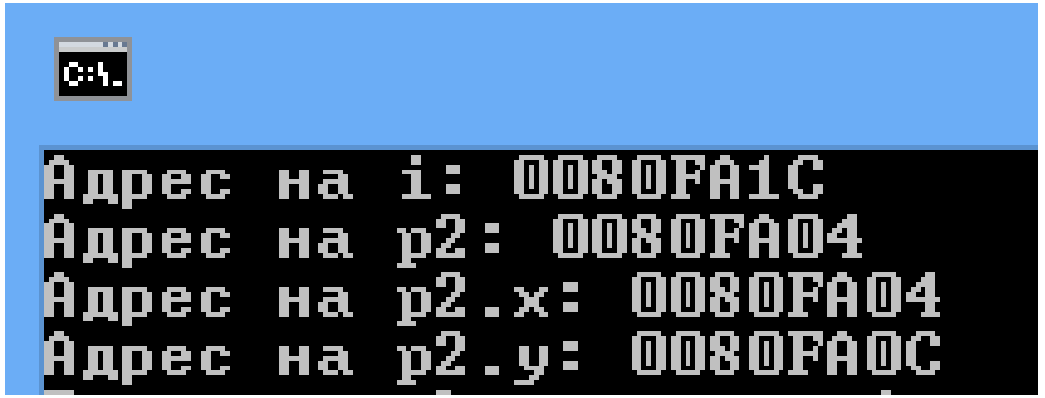
Променливите са статични – намират се в стека.

Адреси на променливи

- Стойността на всяка променлива е записана на конкретен адрес;
- Адресът на променлива се взима с оператор &, който се чете „адрес на“ и се задава пред името на променлива;
- Адресът е 32 битово (4 байта) число, което обикновено се записва в шестнадесетичен вид.

Адреси на променливи – пример и примерен резултат

```
int i = 1;
Point2D p2;
p2.x = 1; p2.y = 2;
```




```
cout << "Адрес на i: " << &i << '\n';
cout << "Адрес на p2: " << &p2 << '\n';
cout << "Адрес на p2.x: " << &p2.x << '\n';
cout << "Адрес на p2.y: " << &p2.y << '\n';
```

Указатели

- Указателите са променливи, които сочат данни от определен тип.
- Реализация на механизма за указване: стойностите на указателите са адреси от паметта.
- **Декларираме променлива за указател** като използваме оператор * пред името на променливата и след типа на обектите, към които желаем да сочи указателя:

  <тип> *<променлива за указател към указания тип>;

- Променливата указател може да получава като стойности адреси само на данни, от зададения за указателя тип.

Декларация на указател - пример

```
int *j;      // j е указател към стойности от тип int
Point2D *p; // p е указател към обекти от тип Point2D
```

- Указателите може да сочат към един елемент или масив от елементи.
- Това не се отразява на декларацията;
- При заделянето на памет се описва броят на елементите, за които се заделя памет.

Задаване на стойност на указател

- Указателят получава като стойност адрес:
  - на статична променлива;
  - на динамичен обект, създаден с оператор new.
- Специална стойност за адрес на указателите е NULL (или 0).
- NULL представя нулевия адрес.
- NULL може да се присвои като стойност на всеки указател.
- NULL означава, че указателят не сочи към конкретни данни.

Заделяне на памет за динамични данни

- С оператори new:

  new <тип>;

  Например,

```
new int;
new Point2D;
```

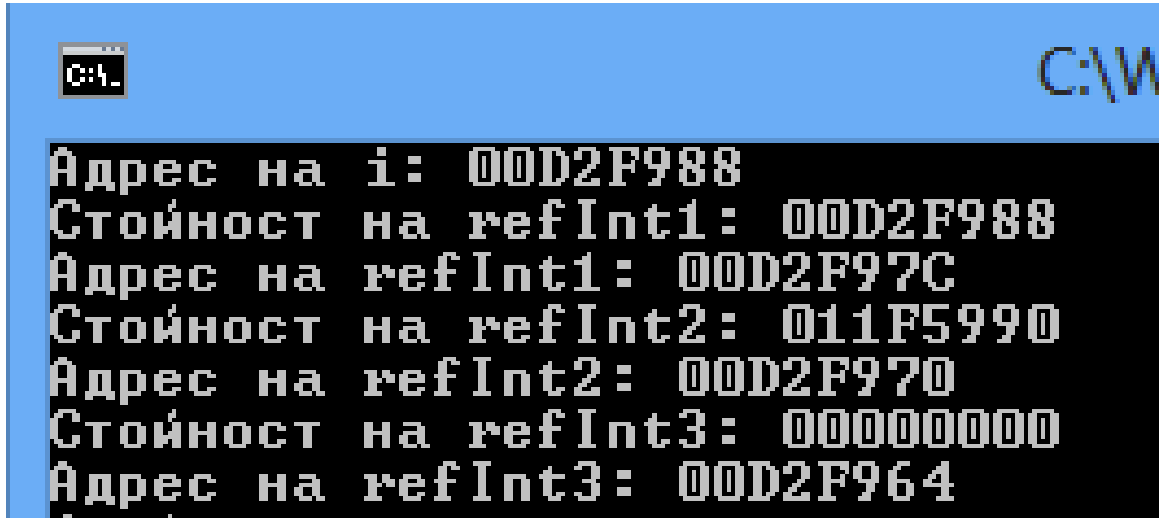
- Операторът new връща адрес от динамичната памет, на който е заделена памет за данни от съответния тип.
- Заделяне на динамична памет за масив от елементи:

  new <тип на елементите>[<брой на елементите>];

```
// заделя се памет за динамичен масив от…
new int[4];     // 4 елемента от тип int
new Point2D[6]; // 6 елемента от тип Point2D
```

Пример: Адреси на променливи и стойности на указатели към примитивен тип

```
int i = 1;    // статична променлива
int *refInt1; // указатели към int
int *refInt2, *refInt3;

refInt1 = &i;      // указател към статична променлива
refInt2 = new int; // указател към динамична стойност – не е зададена стойност в сочената памет
refInt3 = NULL;    // указател към нулевия адрес

// Извеждат се адреси - стойностите на указателите са адреси
cout << "Адрес на i: " << &i << '\n';
cout << "Стойност на refInt1: " << refInt1 << '\n';
cout << "Адрес на refInt1: " << &refInt1 << '\n';
cout << "Стойност на refInt2: " << refInt2 << '\n';
cout << "Адрес на refInt2: " << &refInt2 << '\n';
cout << "Стойност на refInt3: " << refInt3 << '\n';
cout << "Адрес на refInt3: " << &refInt3 << '\n';
```



Обръщение към стойност, сочена от указател. Пример

- С оператор *:

  *<променлива указател>

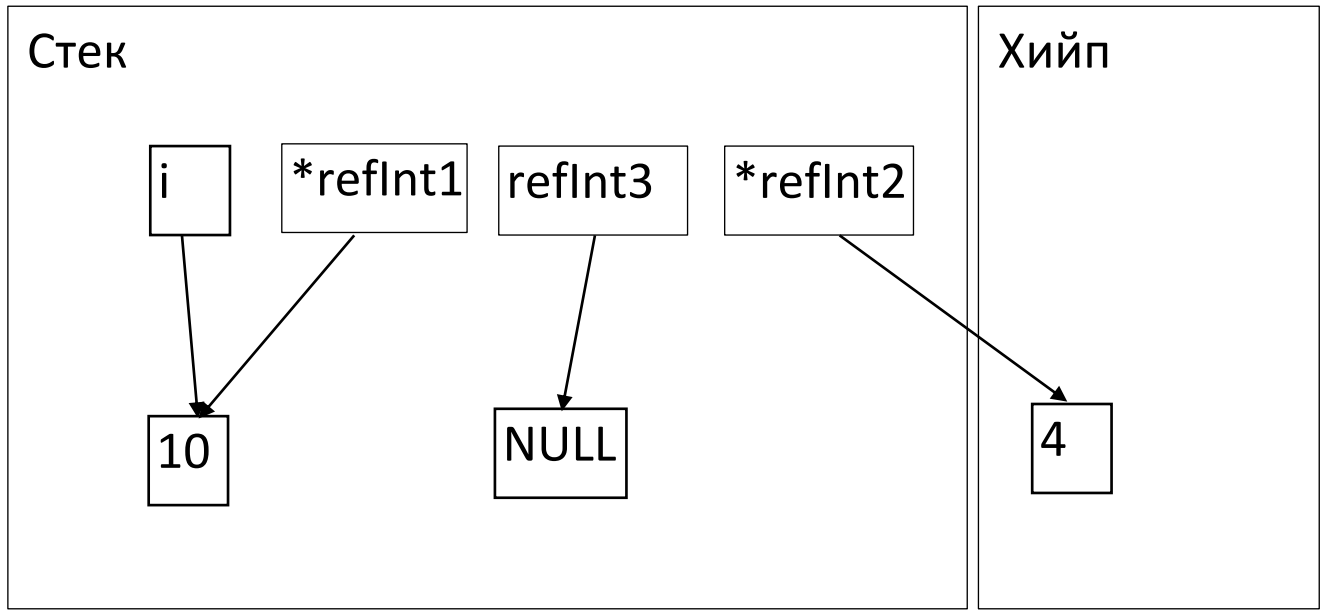
- Обръщението към указател със стойност NULL предизвиква грешка – т.е. при съмнения трябва указателя да се сравнява за NULL:

  ```
  if (refInt3!=NULL){...} или само
  if (refInt3){...}
  ```

- Пример - продължение

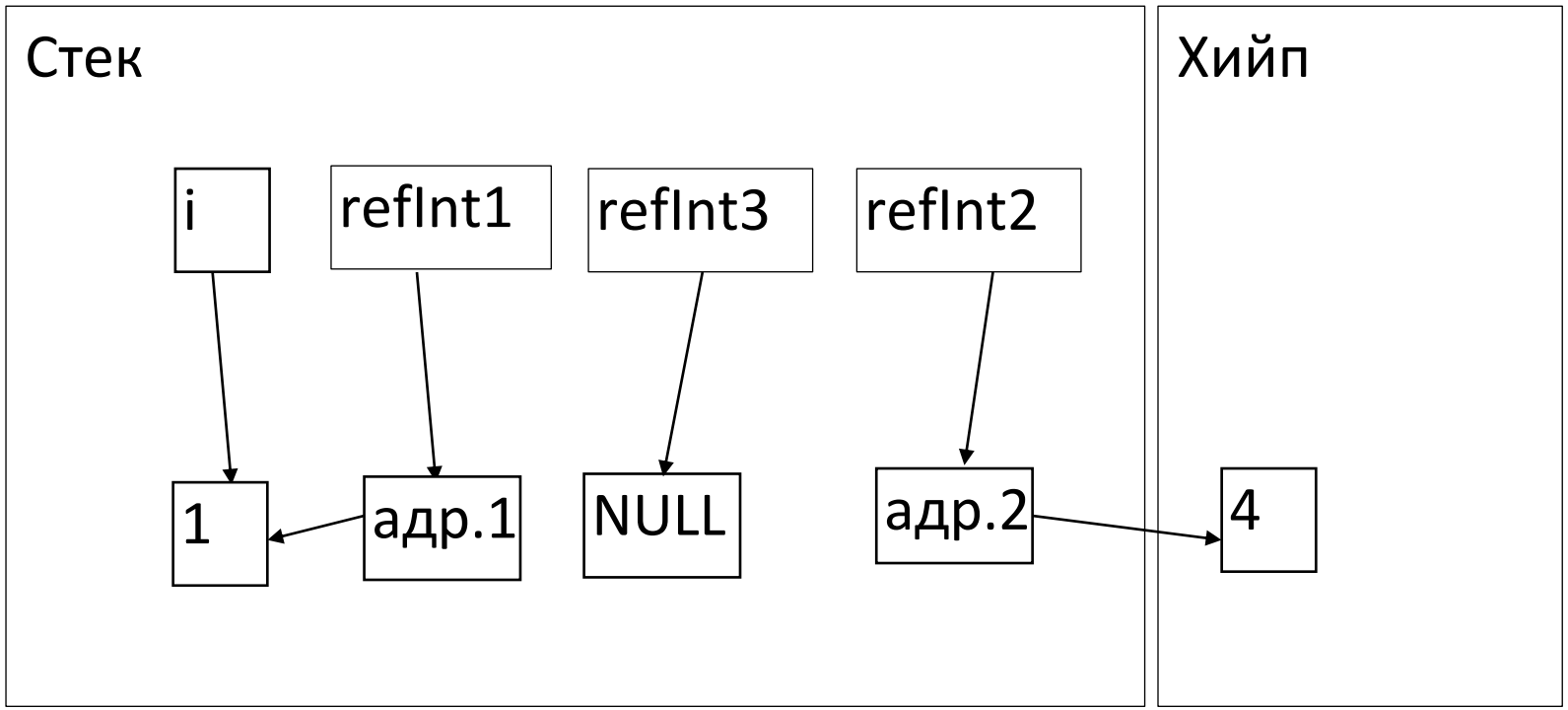
```
*refInt2 = 4; // в паметта сочена от указателя записваме 4

// Извеждат се стойности на числа сочени от указател
cout << "i: " << i << '\n';
cout << "*refInt1: " << *refInt1 << '\n';
cout << "*refInt2: " << *refInt2 << '\n';
//cout << "*refInt3: " << *refInt3 << '\n';// грешка - обръщение към NULL
```

Състояние на променливите в стека и хийпа за разглеждания пример (1)



Указателят refInt1 сочи паметта свързана с променливата i. Променяйки сочената стойност чрез указателя, се променя и стойността на i.

Състояние на променливите в стека и хийпа за разглеждания пример (2)



Стойностите на указателите са адреси, на които се намират данни от съответния тип.

Освобождаване на памет заета от динамични данни - Пример

```
delete refInt2;
// освобождаване на паметта в кода на програмата
// САМО динамична памет сочена от указател
```

Статични и динамични масиви в C++

- Статичните масиви имат константен размер и заемат константна статична памет, а размерът на динамичните масиви може да се определи по време на изпълнение на програмата.
- Декларация на указател към масив (или единична стойност) :

<тип> * <име на променлива - указател>

```
int *arr;
// Деклариране на указател, който ще ползваме за
// обръщение към динамичен масив
```

- Заделяне на памет за масив от цели числа от тип int в динамичната памет:

new <тип на елементите>[<брой на елементите>]

```
int size = 5;
arr = new int[size];
```

- За да достъпваме динамичния масив (в програмата) трябва да го асоциираме с променлива-указател.

Динамични масиви – пример.

Работата с елементи на динамични масиви е почти като при статичните.

```
int main(){
     int size = 5;
     int *arr;  // Деклариране на указател, който ще ползваме за обръщение към динамичен масив
     arr = new int[size];  // операторът new връща адрес от динамичната памет,
                           // на който е създаден динамичния обект (в случая – масив)

     // Обхождането на динамични масиви е като при статични.
     for (int i = 0; i < size; i++) {
          arr[i] = i * 2;
          cout << arr[i] << '\n';
     }

     delete [] arr; // след като динамичния масив стане ненужен, трябва да освободим паметта за него

     return 0;
}
```

Функция за генериране на масив от псевдо-случайни числа

```
#include <iostream>
#include <string>
#include <ctime>
using namespace std;

// Връща масив от size псевдослучайни цели числа от тип int
int* genArray(int size) {       // връща указател към int
    int *arr = new int[size];  // създаване на динамичен масив и указател към него

    srand(time(0));
    for (int i = 0; i < size; i++) {
        arr[i] = rand();         // задаване на стойности на елементите
    }

    return arr;
}
```

Функция за връщане на масив като низ

вече създадена – работи с динамични и статични масиви

```
// Връща масив като низ във вид {el_1, el_2,...el_n}
// Параметри: масив arr и брой на елементите size.
string arrayToString(int arr[], int size) {
    string s = "{";

    for (int i = 0; i < size; i++) {
        s += to_string(arr[i]) + (i<size-1?", ":"");
        // добавя запетая само ако не е достигнат последния елемент
    }

    s += "}";

    return s;
}
```

Използване на функциите за работа с масиви

```
int main(){
    int size = 5;
    int *arr = genArray(size);
    // променливата arr получава като стойност
    // адреса на генерирания от genArray масив

    cout << "Масив: " << arrayToString(arr, size) << '\n';

    delete [] arr;

    return 0;
}
```

Указатели към обекти от съставен тип

- Достъпът до под-елементи на обект, сочен от указател става по два начина:
  - първо се достъпва обекта с оператор * и след това се ползва оператор . (точка);
  - използва се оператор за достъп до под-елемент, на обект сочен от указател:

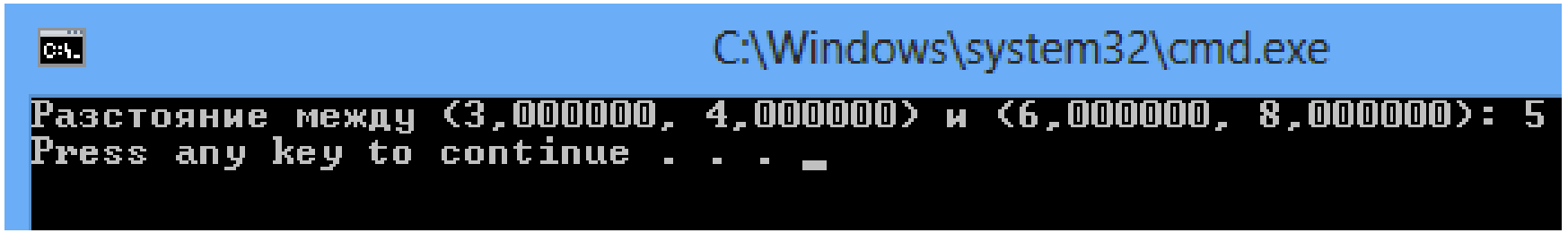
    <указател>-><под-елемент>

Функции за работа с параметри-указатели към съставен тип

```
// Функция, която връща като низ информация за указател към Point2D,
// зададен като параметър.
string pointAsString(Point2D *point) {
    string asString = "(" + to_string(point->x) + ", " + to_string(point->y) + ")";

    return asString;
}
// Функция, която връща разстоянието между две точки от тип указател към Point2D,
// зададени като параметър. Прилага се Питагоровата теорема.
double distance(Point2D *p1, Point2D *p2) {
    double dist = sqrt(pow(p1->x - p2->x, 2) + pow(p1->y - p2->y, 2));

    return dist;
}
```

Използване на функциите

```
int main() {
    setlocale(LC_ALL, "bg");

    Point2D *refP1 = new Point2D;   // указатели към Point2D
    Point2D *refP2 = new Point2D;   // динамични обекти

    (*refP1).x = 3;  // *refP1 - обръщение към стойността на сочения обект
                     // . - обръщение към поделемент на обект
    refP1->y = 4;    // обръщение към поделемент на обект, сочен от указател

    refP2->x = 6;
    refP2->y = 8;

    cout << "Разстояние между " << pointAsString(refP1) << " и " << pointAsString(refP2) << ": “
         << distance(refP1, refP2) << '\n';

    delete refP1
    delete refP2; // освобождаване на паметта за динамичните обекти

    return 0;
}
```



Съставни типове и разположение на обектите в паметта

- Части от сложни обекти може да са разположени едновременно и в статичната и динамичната памет.
- Например, ако създадем статичен обект person за човек

```
struct Person {
     string name;
     Person *boss;
     Point2D location;
} person;
```

  - string е клас, който съхранява символите на низа в динамичната памет (обикновено в динамичен масив от символи);
  - указателят към ръководител (boss) може да сочи динамичен или статичен обект – зависи как програмиста реализира логиката;
  - местоположението се записва в статичната памет.

Псевдоним/Reference (Референция)

- Чрез псевдонимите може да се зададе ново име на СТАТИЧНА променлива.
- Декларацията на псевдоним е:

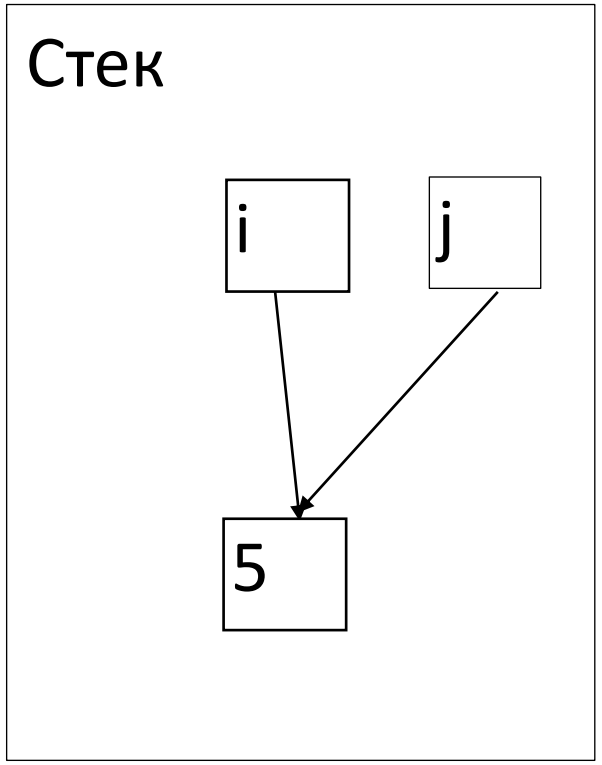
  <тип> &<променлива-псевдоним> = <променлива>;

- Само при декларация на псевдоним се задава съответната му променлива.
- Псевдонимът и съответната му променлива имат една и съща памет за стойността.
- С псевдоним се работи, както с обикновена друга променливи.

Пример за псевдоними

```
int main() {
   int i = 1;
   int &j = i; // j е псевдоним на i -
               // обръщат се към една и съща памет

   j = 5;      // променя се стойността на j (и на i)

   cout << "j: " << j << '\n'; // 5
   cout << "i: " << i << '\n'; // 5

   return 0;
}
```

Променливи от примера в паметта



Псевдоними, указатели и динамични данни (1)

- Въпреки, че реферира статична променлива, псевдонимът може да е свързан и с динамични данни, напр.:

```
int *ukazatel = new int;
int &psevdonim = *ukazatel;
psevdonim = 8;

cout << psevdonim << endl; // Отпечатва се 8
cout << *ukazatel << endl; // Отпечатва се 8
```

Псевдоними, указатели и динамични данни (2)

- **Такива техники за смесване на работа с указатели и псевдоними не са препоръчителни – може да се получат проблеми при последващо използване на псевдонима**:

```
delete &psevdonim; // Освобождава се динамичната памет
cout << psevdonim << endl; // Отпечатва се нещо, напр. -572662307
psevdonim = 7;  // Ще се изпълни, но
                // междувременно паметта може да е заета от друг обект
cout << psevdonim << endl; // Отпечатва се 7
```

- Псевдонимите може да сочат директно и динамични данни

```
int &psevdonim2 = *(new int);
psevdonim2 = 3;
cout << psevdonim2 << endl; // Отпечатва се 3
```

Управление на паметта в други езици за програмиране

- В повечето съвременни езици управлението на паметта е автоматично, а не ръчно както при C и C++, Асемблер и др.
- Такива са Java, C#, PHP, Python, JavaScript и др. При тях:
  - Програмистът няма пряк достъп до адресите (при повечето езици).
  - Не се използват специализирани оператори * и & за работа с адреси.
  - (В PHP се ползва & за предаване по адрес на примитивни типове и масиви. В C# в “unsafe” режим може да се ползват. В езиците може да се вгражда native код от други езици, в които има работа с адреси.)
  - Обикновено логиката е следната:
    - **Данни от примитивен тип се създават в статичната памет:** int i = 10;
    - **Обект от съставен тип се създава в динамичната памет** с оператор new, а променливата, която сочи към него се нарича референция (а не указател):

      Person p = new Person(); // p е референция
  - Достъп до под-елементите на динамични обекти с оператор “.”: p.name
  - Паметта се освобождава автоматично.